A Formal Treatment of Hardware Wallets

A Formal Treatment of Hardware Wallets

Myrto Arapinis[1], Andriana Gkaniatsou[1], Dimitris Karakostas[1,2], and Aggelos Kiayias[1,2]

[1] University of Edinburgh
[2] IOHK
marapini@inf.ed.ac.uk,
{agkaniat,dimitris.karakostas,aggelos.kiayias}@ed.ac.uk

Abstract. Bitcoin, being the most successful cryptocurrency, has been repeatedly attacked with many users losing their funds. The industry's response to securing the user's assets is to offer tamper-resistant hardware wallets. Although such wallets are considered to be the most secure means for managing an account, no formal attempt has been previously done to identify, model and formally verify their properties. This paper provides the first formal model of the Bitcoin hardware wallet operations. We identify the properties and security parameters of a Bitcoin wallet and formally define them in the Universal Composition (UC) Framework. We present a modular treatment of a hardware wallet ecosystem, by realizing the wallet functionality in a hybrid setting defined by a set of protocols. This approach allows us to capture in detail the wallet's components, their interaction and the potential threats. We deduce the wallet's security by proving that it is secure under common cryptographic assumptions, provided that there is no deviation in the protocol execution. Finally, we define the attacks that are successful under a protocol deviation, and analyze the security of commercially available wallets.

1 Introduction

Wallets are the only means to access and manage Bitcoin assets and, although they exist since Bitcoin's inception [20], little or no attention has been paid on formally verifying them. Access to the Bitcoin network, key management, cryptographic operations, and transaction processing are only a few cases of wallet operations. Up until now, there does not exist a specific model of the wallet, nor a thorough threat model, resulting in implementations based on common criteria and security assumptions (*e.g.*, secure key management, correct transaction processing *etc.*) without a complete security treatment. As a result, industry focuses more on securing the cryptographic primitives, and neglects the secure operation of the system as a whole.

The current industry state of the art for managing cryptocurrency assets is hardware wallets. They currently dominate the market as the most secure solution for account management. Although the demand, together with the number of commercially available products, keeps growing, their specifications and security goals remain unclear and understudied. Incorporating expensive hardware

as a wallet is bound to bring some security guarantees; however, proprietary assumptions of the offered functionality and lack of a universal threat model frequently lead to implementations prone to attacks. In this work we formally define the characteristics, specifications and security requirements of hardware wallets in the Universal Composable (UC) Framework [9]; we identify all the potential attack vectors, and the conditions under which a wallet is secure. To that end, we manually inspect the KeepKey, Ledger and Trezor wallets and extract the implementations which we then map to our model. As we show, the wallets are prone to a set of attacks and are secure only under specific assumption. Therefore, our model not only proves the security of existing implementations, but also acts as a reference guide for future implementations.

As wallets are the only way for a user to access her funds, they are repeatedly targeted for attacks that aim to access the account's keys or redirect the payments, ranging from clipboard hijacking [21] and malware [17] to implementation bugs, *e.g.*, the Parity hack in Ethereum [4], and more specific attacks, *e.g.*, brain wallets [26]. In order to address such threats, different ways to harden the wallet's security have been proposed, with the most notable one being the utilization of cryptographic hardware. The module known as a *hardware wallet* is responsible for the account's key management and the execution of the required cryptographic operations. The remaining operations are completed by a dedicated software, either provided together with the hardware or by a third party, with which the hardware communicates. Although hardware wallets are becoming the de facto means of securely managing an account, they have not been formally studied before. Currently, the security of commercially available products can only be checked through manual inspection of their implementation; a process that requires a strong engineering and technical background, and a significant effort and time commitment. Our work aims at bridging the gap between formally modeling and verifying the wallet's properties and claimed specifications. We present a formal model of hardware wallets, which is built using cryptographic primitives and is proven secure under common assumptions. Instead of capturing a hardware wallet as a single module, we conceptualize it as a system of different modules that communicate with each other in order to complete the wallet's operations. This approach allows us to identify a greater set of potential attacks and the conditions for them to be successful. As we show, *perfect cryptographic components by themselves* cannot guarantee security; any module might be proved vulnerable, thus compromising the entire wallet.

Related Work The importance of formal methods for the Bitcoin protocol is well understood, with existing literature showcasing different approaches. Garay *et al.* [12], after extracting and analyzing the core Bitcoin blockchain protocol, presented a formal abstraction to prove that Bitcoin satisfies a set of security and quality properties. Pass *et al.* [22] analyzed the consistency and liveness properties of the consensus protocol in an asynchronous setting, proving Bitcoin secure assuming an upper bound on the network delay. Badertscher *et al.* [6] suggested a universally composable treatment of the Bitcoin ledger, defining Bitcoin's goals and proving that their model is securely realized in the UC frame-

work. Transactions, being a core part of Bitcoin, have also attracted attention. Atzei *et al.* [5] proposed a formal model of Bitcoin transactions in order to prove security *e.g.*, against double-spending attacks, and other blockchain properties *e.g.*, blockchain's decreasing value.

Until now, Bitcoin wallets have only been empirically studied. Previous research on the topic focused on the integrity of transactions and suggested ways to enhance the security of the wallets. Gentilal *et al.* [13] stressed the necessity of separating the wallet into two environments, the trusted and the non-trusted, and proposed that a wallet remains secure against attacks by isolating the sensitive operations in the trusted environment. Similarly, Lim *et al.* [19] and Bamert *et al.* [7], argue that security in Bitcoin wallets equals with tamper-resistance and propose the use of cryptographic hardware. Hardware wallets have not yet been extensively studied, since no formal attempt to specify the functionalities and the security properties of such wallets exists so far. As of September 2018, research has only focused on attacking commercially available implementations. Gkaniatsou *et al.* [14] showed that the low-level communication between the hardware and its client is vulnerable to attacks which escalate to the account management. Their research concluded to a set of attacks on the Ledger wallets, which allowed to take control of the account's funds. Hardware wallets have also been studied against physical attacks. Volotikin [27] showed that specific parts of the Ledger's flash memory are accessible, exposing the private keys used for the second factor verification mechanism. Datko *et al.* presented fault injection, timing and power analysis attacks on KeepKey [1] and Trezor [3], which allowed them to extract the private key.

Our Contributions and Roadmap This work provides a holistic treatment of hardware wallets: from identifying their core specifications and security properties to defining a formal model, which allows reasoning about the offered security of existing wallets and acts as the foundation for designing and implementing new ones. To the best of our knowledge our work is the first to i) define the properties and requirements of hardware wallets, ii) provide a formal model and security guarantees of such wallets, and iii) evaluate the security of commercial products under a formal model.

In Section 2 we define the hardware wallet properties and their security specifications. Section 3.3 presents a formal model for the wallet in the UC framework. We define the ideal functionality of the wallet, which models the wallet's operations and the adversary's capabilities. Instead of conceptualizing the wallet as a single entity, in Section 3.4 we adopt a modular treatment in which the wallet becomes an ecosystem of different components, namely the human, the client and the hardware. Each component runs a protocol, which defines the operations that it carries out, so the wallet functionality is realized as a composition of these protocols. Section 4 addresses the wallet's security. We present the set of attacks that our ideal functionality identifies, including a novel family of attacks that has not been previously discussed. We then prove that the hybrid setting securely realizes the wallet ideal functionality, and showcase examples when per-

fect cryptography is inadequate for securing an account. Finally, we evaluate the security of three commercially available wallets: KeepKey, Ledger, and Trezor.

2 Hardware Wallets

Bitcoin relies on the Elliptic Curve Signature Scheme (ECDSA) for signing the transactions and proving ownership of the assets. An account is defined by a key pair (sk, vk): the public portion vk is hashed to create an address α for receiving assets, and the private portion sk is used to sign transactions that spend the assets that α received. Unauthorized access to sk results in loss of funds, thus raising the issue of securing the account's private keys. A wallet offers access to the Bitcoin network and management of an account, and is either based on software, *i.e.*, is hosted online by a third party or run locally, or hardware.

Threat Model The usage of a broken cryptographic primitive may lead to loss of funds: a broken hash function means potential loss of the receiving funds, whereas a broken signature scheme may result in loss of the spending funds. Protecting against unauthorized access to the wallet's operations has been previously proven to be equally important as using secure cryptographic primitives [8,14]. Hardware wallets are tamper-resistant and offer an isolated environment for the cryptographic primitives. However, if they connect to a compromised client, then any inputs/outputs of the hardware can potentially be malicious. For example, consider Bob, whose account is defined by the key pair (sk, vk), and an adversary $\mathcal{A}$, who is able to forge Bob's signature. In this case any signature $s_{\mathcal{A}}$ of a message $m_{\mathcal{A}}$ chosen by $\mathcal{A}$ can be verified by vk; hence the adversary can spend all assets that Bob has previously received, *i.e.*, the assets sent to the hash of vk. Let us now assume that Bob's signature is unforgeable but $\mathcal{A}$ controls the signing algorithm inputs, such that for any message m that Bob wishes to sign, $\mathcal{A}$ substitutes it with $m_{\mathcal{A}}$. Even though the signature is unforgeable, the adversary can still spend Bob's assets by tampering with the message. Our model captures the family of such attacks, which result in loss of funds by tampering the inputs/outputs of the wallet operations. Thus, the security of a Bitcoin wallet is reduced to the security of the underlying cryptographic primitives *and* the honesty of the communicating parties.

The Wallet Setting Software, not being tamper-resistant, cannot guarantee a secure environment for the wallet's operations. Instead, hardware wallets are designed to offer such an environment by separating the wallet's cryptographic primitives from the other operations *e.g.*, connection to the Bitcoin network. These devices do not offer network connectivity; instead they operate in an offline mode. Due to their limited memory capabilities and the absence of network access, they cannot keep track of the account's activities, *e.g.*, past transactions. Thus, they require connection with a dedicated software, *the client*, which keeps records of the account's actions and provides a usable interface with which the user can interact. Hardware wallets operate under the assumption of a malicious

host, and they provide a trusted path with the user. Both the client and the hardware display transaction related data, which the user compares to decide on their validity. As such, the user becomes part of the system and is responsible for identifying potentially malicious actions of the client.

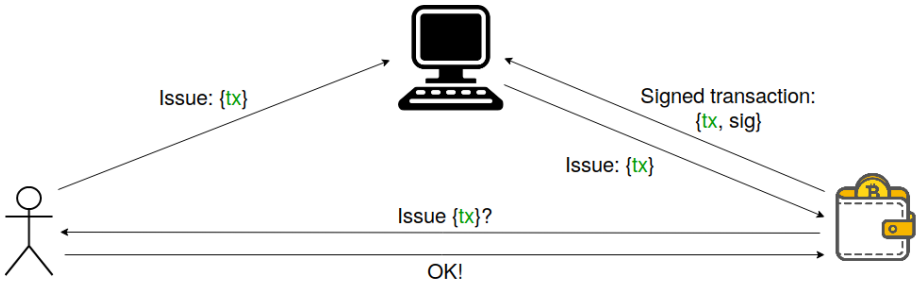

Fig. 1. Transaction issuing in the hardware wallet setting.

The wallet operations are initiated by the user, they are executed by the client, the hardware, or both, and are: i) *Setup*: the hardware generates the master key pair (msk, mvk) and returns the private key msk, *i.e.*, the wallet's *seed*, to the user; currently all wallets are Hierarchical Deterministic, as defined by the BIP32 standard [28], so the keys are derived from a master key pair by the simplified functions $sk_i = msk + hash(i, mvk) (\text{mod } n)$ and $vk_i = mvk + hash(i, mvk) \times N$, where i is the index of the key and n, N are public parameters of the used Elliptic Curve; ii) *Session Initialization*: the hardware connects to the client and sends it the master public key mvk; iii) *Generate Address*: both the client and the hardware generate a new address and return it to the user. The hardware derives from (msk, mvk) a new pair (sk_i, vk_i), generates a new address α_i and returns it to the user. The client may either generate vk_i using mvk or receive vk_i from the hardware, then generates and stores the corresponding address, and finally returns it to the user; iv) *Calculate Balance*: given a list of the account's addresses, the client iterates over the ledger's transactions, calculates the account's available assets and returns this amount to the user; v) *Transaction Issuing*: the user provides the payment data to the client, which then forwards them to the hardware together with the available inputs, *i.e.*, the account's addresses and balances, and requests its signature. The hardware checks whether the input addresses belong to the managed account and generates a *change address* upon demand, *i.e.*, if the balance is larger than the payable amount plus transaction fees. Then, it requests the user's approval of the payment data. If the user confirms the payment, then the hardware signs it and returns the signature together with the corresponding public key to the client in order to publish it. Figure 1 presents an abstraction of the transaction issuing process in the hardware-enhanced wallet setting.

In our model a wallet is not a single module, but rather an "ecosystem", which consists of different modules that communicate during an operation: the user, the client and the hardware. In order to treat it under the UC framework, we will describe an ideal functionality for it as well as its real world implementation. In the ideal world the wallet is a single component (functionality) responsible for all the aforementioned operations while in reality the wallet is split into multiple modules which communicate during the execution of each operation. In the real world, the wallet emerges through the interaction of the human operator, the client (which is a device like a desktop computer or tablet/smartphone) and a tamper resistant hardware component.

Ideal World The wallet functionality, $\mathcal{F}_{\text{w}}$, is responsible for the wallet's operations. $\mathcal{F}_{\text{w}}$ interacts with the global Bitcoin ledger functionality $\mathcal{G}_{\mathsf{LEDGER}}$, as defined in [6], in order to execute operations requiring access to the decentralized system. $\mathcal{G}_{\mathsf{LEDGER}}$ is the ideal functionality that models the Bitcoin ledger and allows a wallet to register itself, publish transactions and retrieve the state of the ledger, *i.e.*, all published transactions. $\mathcal{F}_{\text{w}}$ generates a unique address per public key and also incorporates a signature functionality, $\mathcal{F}_{\mathsf{SIG}}$ as defined in [10] (for convenience we will treat $\mathcal{F}_{\mathsf{SIG}}$ as a separate component in the ideal world). The wallet registers itself with $\mathcal{F}_{\mathsf{SIG}}$ which creates fresh keys for the account upon request, *e.g.*, during address generation, and signs messages, *e.g.*, transactions. $\mathcal{F}_{\mathsf{SIG}}$ is also accessed by the validation predicate of $\mathcal{G}_{\mathsf{LEDGER}}$ in order to verify a transaction's signature during the validation stage.

Real World The operations are executed by a set of communicating parties: the *hardware*, the *client* and the *user*. Thus the protocols of the hardware π_{hw}, the client π_{client}, and the human π_{human} define the actions of the corresponding parties. The hardware protocol π_{hw}, uses a signature scheme $\Sigma \equiv \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$, a cryptographic hash function H and a pseudorandom key generation function $\mathsf{HierarchicalKeyGen}(msk, i)$, in order to derive children keys from the master key. A basic assumption of this setting is that π_{client} runs in an untrusted environment, *i.e.*, we do not consider the software to be secure. Thus, connection to a malicious client, in our model, is equivalent to corruption of the client by the adversary. The human communicates with the hardware and the client via a secure channel, *i.e.*, the user interacts directly with the device.

Figure 2 presents the ideal and the real world settings. In both worlds the environment Z interacts with the adversary, *i.e.*, in the ideal world it interacts with the simulator S, and in the real world with the adversary $\mathcal{A}$. In the ideal world the wallet consists of the ideal wallet functionality $\mathcal{F}_{\text{w}}$ and the signature functionality $\mathcal{F}_{\mathsf{SIG}}$; in the real world it consists of the combination of the user, client and hardware wallet parties who execute the respective protocols $(\pi_{human}, \pi_{client}, \pi_{hw})$. The communication between the human, the client and the hardware is achieved over a *UC-secure channel protocol* as presented by Canetti [11]: the adversary is able to observe the encrypted communication between the honest parties and only retrieve the length of the exchanged messages. In practice, this can be achieved by establishing a secure channel between the

client and the hardware module using standard key exchange techniques, while the human-hardware channel is assumed to be secure by default. In the absence of a secure channel, the adversary may tamper with the communication thus, in our model, an insecure channel is equivalent to the client being corrupted.

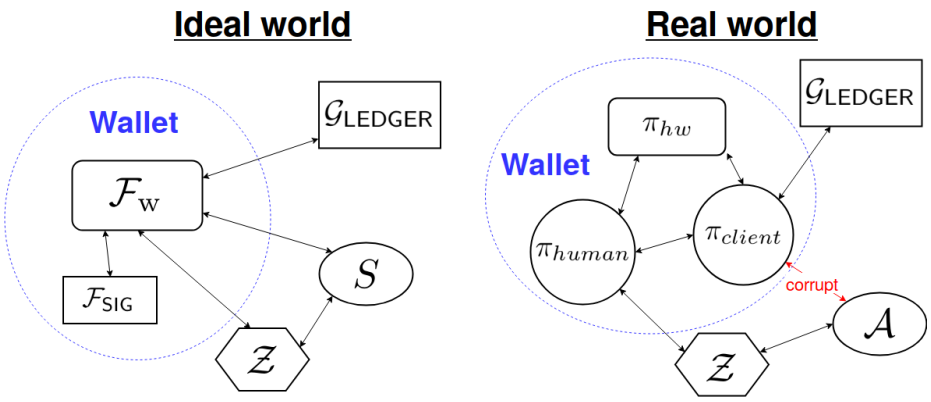

Fig. 2. A high-level comparison of the ideal and the real world.

3 Formal Model

This section defines a formal model of the hardware wallet ecosystem in the UC Framework, in which we compare the execution of a security definition, *i.e.*, the ideal setting, with a concrete protocol setting, *i.e.*, the real world. We first define a wrapper for the validation predicate which is used by the Bitcoin ledger functionality $\mathcal{G}_{\text{LEDGER}}$ and is accessed in both the ideal and the real worlds upon publishing a transaction. In the ideal world, a functionality defines the wallet operations. In the real world, the hardware wallet is defined as a hybrid setting which we prove to securely realize the ideal definition.

3.1 Notations

An address α is a unique string chosen from $\{0,1\}^{\ell}$, where ℓ is the length of α in bits, and is associated with a payment key pair (vk, sk), where vk is the public and sk the private key. The wallet's key pairs and, consequently, the addresses are generated using the master key pair (msk, mvk), which is randomly selected from the key domain $\mathbb{K}$ upon the wallet's setup. A transaction is defined as a tuple $tx := (\alpha_{\text{s}}, \alpha_{\text{r}}, \theta_{\text{pay}}, \alpha_{\text{c}}, \theta_{\text{change}})$, where α_{s} denotes the sender's address, α_{r} the receiver's address and α_{c} the change address; θ_{pay}, θ_{change} θ_{fee} are the payment, change and fee funds respectively, where θ_{change} equals to the account's

balance minus the payment and the fee amounts, *i.e.*, $\theta_{\text{change}} := \text{balanceOf}(\alpha_s) - \theta_{\text{pay}} - \theta_{\text{fee}}$. A signed transaction is the tuple (tx, vk, σ), where σ is the signature of tx under the public key vk. We note that, for ease of notation, this is a simplified transaction model - adapting it for multiple inputs and outputs, *e.g.*, to properly model Bitcoin, should be straightforward though. The parties that execute an operation are the user $\mathcal{U}$ *i.e.*, the owner of the wallet, the client to which the hardware connects $\mathcal{C}$, and the hardware $\mathcal{H}$. Each message is associated with a session id $sid' = \mathcal{UCH}$, which defines the parties that the message is related with.

3.2 Validation Predicate

The Bitcoin Ledger functionality $\mathcal{G}_{\mathsf{LEDGER}}$, as defined in [6], is parameterized with the validation predicate Validate. This predicate identifies whether a transaction can be added to the buffer, *i.e.*, whether it is valid for publishing to the ledger. Although a concrete instantiation is not provided, it is stated that it takes as input the candidate transaction, the buffer and the current state. The candidate transaction consists of the signed transaction $stx = (tx, vk, \sigma)$ and the ledger parameters, *e.g.*, the transaction id, which is a unique identifier of the transaction, and the timestamp, *i.e.*, the time which is defined by a global clock. Intuitively, in Bitcoin the state is the blockchain and the buffer is the mempool which contains the transactions that have not yet been included in a block. The validation predicate is used for the signature verification of a candidate transaction. This is formalized with a wrapper ValidateWrapper, which wraps all instantiations of the validation predicate. In the ideal world the wrapper accesses the signature functionality $\mathcal{F}_{\mathsf{SIG}}$ to verify the transaction's signature: if the signature is not valid, then it directly outputs 0, otherwise it performs all additional checks, such as verifying the funds which are consumed and checking whether the amounts are valid. We refrain from constraining our setting to a specific Validate predicate, but rather describe it for all generic ledger settings. The ideal wrapper IdealValidateWrapper is described in Algorithm 1. The real world wrapper RealValidateWrapper uses a signature scheme instead of $\mathcal{F}_{\mathsf{SIG}}$ and behaves similarly to Algorithm 1, *i.e.*, it first parses BTX and then performs the same branch checks on $\mathsf{Verify}(tx, vk, \sigma)$ and returns the proper boolean value.

Algorithm 1 The validation predicate wrapper, parameterized by Validate and $\mathcal{F}_{\mathsf{SIG}}$. The input is a transaction BTX, the buffer buffer and the state state.

```
function IdealValidateWrapper(BTX, buffer, state)
    (tx, vk, σ, txid, τ_L, p_i) := parse(BTX)
    Send (VERIFY, sid, tx, σ, vk) to F_SIG and receive (VERIFIED, sid, tx, f)
    if f = 0 then
        return 0
    else
        return Validate(BTX, buffer, state)
    end if
end function
```

3.3 The Wallet Ideal Model

$\mathcal{F}_{\text{w}}$ incorporates $\mathcal{F}_{\mathsf{SIG}}$ and runs in the $\mathcal{G}_{\mathsf{LEDGER}}$-setting, interacting with the adversary $\mathcal{A}$, a set of parties $\mathbb{P}$ and the environment $\mathcal{Z}$, and keeps the initially empty items: i) $A_{[]}$: a list of lists of addresses and the corresponding public keys, (α, vk), ii) $B_{[]}$: a list of lists of the account's addresses and their corresponding balance, (α, θ), and iii) $K_{[]}$: a list of master key pairs (mvk, msk). $\mathcal{F}_{\text{w}}$ realizes the following operations: i) *Wallet setup*: Upon a setup request, it initializes the list of addresses, generates the account's master key pair, registers to $\mathcal{G}_{\mathsf{LEDGER}}$ and returns the master private key. ii) *Client Corruption*: When $\mathcal{A}$ corrupts a client $\mathcal{C}$, $\mathcal{F}_{\text{w}}$ leaks the past public keys and addresses that $\mathcal{C}$ has obtained. iii) *Client session initialization*: In order to start a new session, it identifies the $\mathcal{C}$ defined in sid' and and returns a new assigned pass phrase "pass"; in the real world, the pass phrase acts as the authentication mechanism between the parties. iv) *Address generation*: It requests a new public key from $\mathcal{F}_{\mathsf{SIG}}$ and picks an associated address at random. It then stores the new address in the corresponding list and also returns it to $\mathcal{Z}$. If the connected client is corrupted, then the functionality leaks the address and the public key to $\mathcal{A}$. v) *Balance calculation*: If $\mathcal{C}$ is honest then it queries the ledger to retrieve the blockchain; if the connected client is corrupted, then it requests from $\mathcal{A}$ to provide the chain. Then, it calculates the amount of available assets and returns it to $\mathcal{Z}$. vi) *Transaction issuing*: Upon receiving a transaction request, if $\mathcal{C}$ is corrupted, then it leaks the transaction information to the adversary and retrieves a new transaction object from it. If $\mathcal{U}$ is also corrupted, then it discards the original request and keeps the adversarial transaction, otherwise it ignores the adversary's response. Finally, it requests a signature from $\mathcal{F}_{\mathsf{SIG}}$ for the transaction which it then publishes to $\mathcal{G}_{\mathsf{LEDGER}}$.

Functionality $\mathcal{F}_{\text{w}}$

All messages below contain a session id of the form $\mathsf{sid} = (\mathbb{P}, \mathsf{sid}')$.

- **Setup:** Upon receiving $(\textsc{Setup}, \mathsf{sid})$ from some party $\mathcal{U} \in \mathbb{P}$, forward it to $\mathcal{A}$. Then add the empty list $A_{\mathcal{U}}$ to $A_{[]}$, register with $\mathcal{G}_{\mathsf{LEDGER}}$, pick the master key pair $(msk_{\mathcal{U}}, mvk_{\mathcal{U}}) \xleftarrow{\$} \mathbb{K}$ and add it to $K_{[]}$ and return $(\textsc{SetupOK}, \mathsf{sid})$ to $\mathcal{U}$.
- **Client Corruption:** When $\mathcal{A}$ corrupts a party $\mathcal{C}$, for every $\mathcal{U}$ such that a **Setup** session with $\mathcal{C}$ has been completed send $(\textsc{AddressList}, \mathsf{sid}, A_{\mathcal{U}})$ and $(\textsc{MasterPubKey}, \mathsf{sid}, mvk_{\mathcal{U}})$ to $\mathcal{A}$.
- **Initialize Client Session:** Upon receiving $(\textsc{InitSession}, \mathsf{sid})$ from party $\mathcal{U}$, pick $pass_{client} \xleftarrow{\$} \{0,1\}^{\lambda}$ and send $(\textsc{InitSession}, \mathsf{sid}, pass_{client})$ to $\mathcal{C}$. If $\mathcal{C}$ is corrupted, then send $(\textsc{InitSession}, \mathsf{sid}, pass_{client})$ to $\mathcal{A}$ and wait for a response $(\textsc{InitSessionOK}, \mathsf{sid}, pass_{client})$. Finally, send $(\textsc{Session}, \mathsf{sid}, pass_{client})$ to $\mathcal{U}$.
- **Generate Address:** Upon receiving $(\textsc{GenAddr}, \mathsf{sid})$ from $\mathcal{U}$, send $(\textsc{KeyGen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{SIG}}$. Upon receiving $(\textsc{Verification Key}, \mathsf{sid}, vk)$ from $\mathcal{F}_{\mathsf{SIG}}$, pick an address $\alpha \xleftarrow{\$} \{0,1\}^{\ell}$ and add (α, vk) to $A_{\mathcal{U}}$. If $\mathcal{C}$

is corrupted then send $(\text{ADDRESS}, \mathsf{sid}, (\alpha, vk))$ to $\mathcal{A}$ and wait for a response $(\text{ADDRESSOK}, \mathsf{sid}, \alpha')$. If $\mathcal{U}$ is corrupted then set a $:= \alpha'$, else set a $:= \alpha$. Finally, return $(\text{ADDRESS}, \mathsf{sid}, \text{a})$ to $\mathcal{U}$.

- **Calculate Balance:** Upon receiving $(\text{GETBALANCE}, \mathsf{sid})$ from $\mathcal{U}$, send $(\text{READ}, \mathsf{sid})$ to $\mathcal{G}_{\mathsf{LEDGER}}$ and wait for the response $(\text{READ}, \mathsf{sid}, \mathsf{chain})$. If $\mathcal{C}$ is corrupted, then send $(\text{READ}, \mathsf{sid})$ to $\mathcal{A}$ and, upon receiving the response $(\text{READ}, \mathsf{sid}, \mathsf{chain}')$, set $\mathsf{chain} := \mathsf{chain}'$. Then set $\mathsf{balance} := 0$, initialize the list $B_{\mathcal{U}} \in B_{[]}$ which contains $(\text{a}, 0)$ for every address $(\text{a}, \cdot)$ in $A_{\mathcal{U}}$, and $\forall tx \in \mathsf{chain}$, *i.e.*, the ordered transactions in the ledger such that $tx = (\alpha_{\text{s}}, \alpha_{\text{r}}, \theta_{\text{pay}}, \alpha_{\text{c}}, \theta_{\text{change}})$, do:
  - If $\exists(\alpha_{\text{s}}, \cdot) \in A_{\mathcal{U}}$, then update the entry $(\alpha_{\text{s}}, \theta_{\text{past}}) \in B_{\mathcal{U}}$ to $(\alpha_{\text{s}}, 0)$;
  - If $\exists(\alpha_{\text{r}}, \cdot) \in A_{\mathcal{U}}$, then update the entry $(\alpha_{\text{r}}, \theta_{\text{past}}) \in B_{\mathcal{U}}$ to $(\alpha_{\text{r}}, \theta_{\text{past}} + \theta_{\text{pay}})$;
  - If $\exists(\alpha_{\text{c}}, \cdot) \in A_{\mathcal{U}}$, then update the entry $(\alpha_{\text{c}}, \theta_{\text{past}}) \in B_{\mathcal{U}}$ to $(\alpha_{\text{c}}, \theta_{\text{past}} + \theta_{\text{change}})$;

  Finally, for every $(\cdot, \theta) \in B_{\mathcal{U}}$ do $\mathsf{balance} := \mathsf{balance} + \theta$ and send $(\text{BALANCE}, \mathsf{sid}, \mathsf{balance})$ to $\mathcal{U}$.
- **Issue Transaction:** Upon receiving $(\text{ISSUETX}, \mathsf{sid}, (\alpha_{\text{r}}, \theta_{\text{pay}}, \theta_{\text{fee}}))$ from $\mathcal{U}$, if $\mathcal{C}$ is corrupted then forward the message to $\mathcal{A}$ and wait for a response $(\text{ISSUETX}, \mathsf{sid}, pass_{client}, (\alpha'_{\text{r}}, \theta'_{\text{pay}}, \theta'_{\text{fee}}))$. If $\mathcal{U}$ is corrupted then set $(\alpha_{\text{r}}, \theta_{\text{pay}}, \theta_{\text{fee}}) := (\alpha'_{\text{r}}, \theta'_{\text{pay}}, \theta'_{\text{fee}})$. Then find $(\alpha_{in}, \theta_{in}) \in B_{\mathcal{U}} : \theta_{in} \geq \theta_{\text{pay}} + \theta_{\text{fee}}$. If such entry exists then compute an address α_{c} and its public key vk_{c} as per the *Generate Address* interface, set $\theta_{\text{change}} := \theta_{in} - \theta_{\text{pay}} - \theta_{\text{fee}}$ and $tx := (\alpha_{in}, \alpha_{out}, \theta_{\text{pay}}, \alpha_{\text{c}}, \theta_{\text{change}})$, send $(\text{SIGN}, \mathsf{sid}, tx)$ to $\mathcal{F}_{\mathsf{SIG}}$ and wait for $(\text{SIGNATURE}, \mathsf{sid}, tx, \sigma)$. Then find $(\alpha_{in}, vk) \in A_{\mathcal{U}}$ and set $stx := (tx, vk, \sigma)$. If $\mathcal{C}$ is corrupted, send $(\text{ADDRESS}, \mathsf{sid}, \alpha_{\text{c}}, vk_{\text{c}})$ and $(\text{SUBMIT}, \mathsf{sid}, stx)$ to $\mathcal{A}$ and wait for the response $(\text{SUBMITOK}, \mathsf{sid})$. Finally, send $(\text{SUBMIT}, \mathsf{sid}, stx)$ to $\mathcal{G}_{\mathsf{LEDGER}}$.

3.4 The Hardware Wallet Hybrid Setting

The hybrid setting consists of the human π_{human}, client π_{client}, and hardware π_{hw} protocols, which define the set of operations run by the parties.

Human Protocol π_{human} interacts with $\mathcal{C}$, $\mathcal{H}$, and the environment $\mathcal{Z}$, and defines the following, initially empty, items: i) T: a list of transactions $tx = (\alpha_r, \theta_{pay}, \theta_{fee})$, and ii) S: a list of client sessions *sid*. The model assumes that a session is initialized when $\mathcal{U}$ connects the hardware module to the client device and assigns a pass phrase $pass_{client} \in \{0,1\}^{\lambda}$ to each client with which she interacts, which is chosen at random upon session initialization. Although it is assumed that the user samples an unguessable pass phrase, future work will explore functionalities that allow a malicious client to perform (dictionary) password attacks against it. Also $\mathcal{U}$ keeps track of the initiated sessions and pending transactions. The user does not perform complex computations, *e.g.*, verifying a signature, or maintain a large state, like the entire list of generated addresses. It

is only assumed to have a memory T, only as large as the pending transactions it is processing, and also that it is capable of performing equality checks between strings.

Protocol π_{human}

- **Setup:** Upon receiving $(\text{SETUP}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{H}$, and initialize T to empty. Then upon receiving $(\text{SETUPOK}, \mathsf{sid})$ from $\mathcal{H}$ forward it to $\mathcal{Z}$.
- **Initialize Client Session:** Upon receiving $(\text{INITSESSION}, \mathsf{sid})$ from $\mathcal{Z}$, pick $pass_{client} \xleftarrow{\$} \{0,1\}^{\lambda}$ and send $(\text{INITSESSION}, \mathsf{sid}, pass_{client})$ to $\mathcal{C}$. Upon receiving $(\text{INITSESSION}, \mathsf{sid}, pass'_{client})$ from $\mathcal{H}$, if $pass'_{client} = pass_{client}$ then add $pass_{client}$ to S and send $(\text{SESSION}, \mathsf{sid}, pass'_{client})$ to $\mathcal{H}$ and to $\mathcal{Z}$.
- **Generate Address:** Upon receiving $(\text{GENADDR}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{C}$ and wait for two messages, $(\text{ADDRESS}, \mathsf{sid}, \alpha_{client})$ from $\mathcal{C}$ and $(\text{ADDRESS}, \mathsf{sid}, \alpha_{hw})$ from $\mathcal{H}$. Upon receiving them, if $\alpha_{client} = \alpha_{hw}$ then send $(\text{ADDRESS}, \mathsf{sid}, \alpha_{hw})$ to $\mathcal{Z}$.
- **Calculate Balance:** Upon receiving $(\text{GETBALANCE}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{C}$. Then upon receiving $(\text{BALANCE}, \mathsf{sid}, \mathsf{balance})$ from $\mathcal{C}$, forward it to $\mathcal{Z}$.
- **Issue Transaction:** Upon receiving $(\text{ISSUETX}, \mathsf{sid}, tx)$ from $\mathcal{Z}$, such that $tx = (\alpha_r, \theta_{pay}, \theta_{fee})$, add tx to T and forward the message to $\mathcal{C}$. Upon receiving $(\text{CHECKTX}, \mathsf{sid}, pass_{client}, tx', \mathsf{balance}')$ from $\mathcal{H}$, if $pass_{client} \in S$, $tx' \in T$ and $\mathsf{balance}' = \mathsf{balance} - \theta_{pay} - \theta_{fee}$ then remove tx' from T and send $(\text{ISSUETX}, \mathsf{sid}, pass_{client}, tx)$ to $\mathcal{H}$.

Client Protocol The client $\mathcal{C}$ interacts with the user $\mathcal{U}$, the hardware wallet $\mathcal{H}$ and the environment $\mathcal{Z}$. The protocol π_{client} defines the following items: i) mvk: the master public key of the wallet, ii) i: the key derivation index, iii) $pass$: the pass phrase that the user assigns to the client, iv) A_{client}: a list of the account's addresses, and v) T_{utxo}: a list of unspent balances like $tx = (\alpha_{in}, \theta_{in})$, where $\alpha_{in} \in A_{client}$ and $\theta_{in} > 0$. $\mathcal{C}$ acts a proxy between $\mathcal{U}$ and $\mathcal{H}$, provides connectivity to the ledger and executes blockchain-related operations, *e.g.*, computing the account's balance. Although during the address generation, $\mathcal{C}$ retrieves the public key from $\mathcal{H}$, in practice this is optional and the client can generate the address independently via the derivation process of the hierarchical deterministic wallets.

Protocol π_{client}

- **Initialize Client Session:** Upon receiving $(\text{INITSESSION}, \mathsf{sid}, pass_{client})$ from $\mathcal{U}$, forward it to $\mathcal{H}$. Upon receiving $(\text{MASTERPUBKEY}, \mathsf{sid}, mvk)$ from $\mathcal{H}$, set $pass := pass_{client}$, $\mathsf{mvk} := mvk$ and $i := 1$.
- **Generate Address:** Upon receiving $(\text{GENADDR}, \mathsf{sid})$ from $\mathcal{U}$, forward it to $\mathcal{H}$. Then upon receiving $(\text{PUBKEY}, \mathsf{sid}, vk_i)$ from $\mathcal{H}$, compute $\alpha_i :=$

$\mathsf{H}(vk_i)$, set $i := i+1$ and add α_i to A_{client}. Finally, send $(\textsc{Address}, \mathsf{sid}, \alpha_i)$ to $\mathcal{U}$.

- **Calculate Balance:** Upon receiving $(\textsc{GetBalance}, \mathsf{sid})$ from $\mathcal{U}$, send $(\textsc{Read}, \mathsf{sid})$ to $\mathcal{G}_{\mathsf{LEDGER}}$. Upon receiving $(\textsc{Read}, \mathsf{sid}, \mathsf{chain})$ from $\mathcal{G}_{\mathsf{LEDGER}}$, set $\mathsf{balance} := 0$ and T_{utxo} to the empty list and $\forall tx \in \mathsf{chain}$, *i.e.*, the ordered transactions in the ledger such that $tx = (\alpha_s, \alpha_r, \theta_{pay}, \alpha_c, \theta_{change})$, do:
  - If $\alpha_s \in A_{client}$ then update the entry $(\alpha_s, \theta_{past}) \in T_{utxo}$ to $(\alpha_s, 0)$;
  - If $\alpha_r \in A_{client}$ then update the entry $(\alpha_r, \theta_{past}) \in T_{utxo}$ to $(\alpha_r, \theta_{past} + \theta_{pay})$;
  - If $\alpha_c \in A_{client}$ then update the entry $(\alpha_c, \theta_{past}) \in T_{utxo}$ to $(\alpha_c, \theta_{past} + \theta_{change})$;

  Finally, for every $(\cdot, \theta) \in T_{utxo}$ do $\mathsf{balance} := \mathsf{balance} + \theta$ and send $(\textsc{Balance}, \mathsf{sid}, \mathsf{balance})$ to $\mathcal{U}$.
- **Issue Transaction:** Upon receiving $(\textsc{IssueTX}, \mathsf{sid}, tx)$ from $\mathcal{U}$, such that $tx = (\alpha_r, \theta_{pay}, \theta_{fee})$, send $(\textsc{SignTx}, \mathsf{sid}, pass, tx, T_{utxo})$ to $\mathcal{H}$. Upon receiving $(\textsc{ChangeIndex}, \mathsf{sid}, idx)$, set $i := idx$ and compute and store the public key and the address for the change as in the *Generate Address* interface. Then upon receiving $(\textsc{SignTx}, \mathsf{sid}, stx)$ from $\mathcal{H}$, send $(\textsc{Submit}, \mathsf{sid}, stx)$ to $\mathcal{G}_{\mathsf{LEDGER}}$.

Hardware Wallet Protocol The hardware $\mathcal{H}$ interacts with $\mathcal{C}$ and $\mathcal{U}$ and runs the protocol π_{hw}, which defines the following items: i) i: the key derivation index, ii) S: a list of the active client sessions, iii) (msk, mvk): the master key pair of the wallet, and iv) A: a list that contains tuples like $(i, \alpha_i, sk_i, vk_i)$ where i is an index, α_i a generated address and (sk_i, vk_i) the corresponding key pair.

Protocol π_{hw}

- **Setup:** Upon receiving $(\textsc{Setup}, \mathsf{sid})$ from $\mathcal{U}$, initialize S and A to empty lists. Then compute $(msk, mvk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and set $i := 1$. Finally, return $(\textsc{Setup}, \mathsf{sid}, msk)$ to $\mathcal{U}$.
- **Initialize Client Session:** Upon receiving $(\textsc{InitSession}, \mathsf{sid}, pass_{client})$ from $\mathcal{C}$, forward it to $\mathcal{U}$. Upon receiving $(\textsc{Session}, \mathsf{sid}, pass'_{client})$ from $\mathcal{U}$ add $pass'_{client}$ to S and send $(\textsc{MasterPubKey}, \mathsf{sid}, mvk)$ to $\mathcal{C}$.
- **Generate Address:** Upon receiving $(\textsc{GenAddr}, \mathsf{sid})$ from $\mathcal{C}$, compute $(sk_i, vk_i) := \mathsf{HierarchicalKeyGen}(msk, i)$ and $\alpha_i := \mathsf{H}(vk_i)$. Then store $(i, \alpha_i, sk_i, vk_i)$ to A, set $i := i + 1$, and return $(\textsc{Address}, \mathsf{sid}, \alpha_i)$ to $\mathcal{U}$ and $(\textsc{PubKey}, \mathsf{sid}, vk_i)$ to $\mathcal{C}$.
- **Issue Transaction:** Upon receiving $(\textsc{SignTx}, \mathsf{sid}, pass_{client}, tx, T_{utxo})$ from $\mathcal{C}$, where $tx = (\alpha_r, \theta_{pay}, \theta_{fee})$, find an entry $(\alpha_{in}, \theta_{in}) \in T_{utxo}$: $\theta_{in} \geq \theta_{pay} + \theta_{fee}$. If such entry exists, then: i) find $(\cdot, \alpha_{in}, sk_{in}, vk_{in}) \in A$, ii) compute the remaining change $\theta_{change} := \theta_{in} - \theta_{pay} - \theta_{fee}$, iii) create a change address α_c as in the *Generate Address* interface, and

iv) compute $\mathsf{balance}$ as the sum of θ for every $(\cdot, \theta) \in T_{utxo}$ and set $\mathsf{balance}' := \mathsf{balance} - \theta_{pay} - \theta_{fee}$. Then send $(\textsc{ChangeIndex}, \mathsf{sid}, i)$ to $\mathcal{C}$ and $(\textsc{CheckTx}, \mathsf{sid}, pass_{client}, tx', \mathsf{balance}')$ to $\mathcal{U}$, where $tx' = (\alpha_r, \theta_{pay}, \theta'_{fee})$. Upon receiving $(\textsc{IssueTx}, \mathsf{sid}, pass_{client}, tx)$ from $\mathcal{U}$, set $tx := (\alpha_{in}, \alpha_r, \theta_{pay}, \alpha_c, \theta_{change})$, compute $stx := (tx, vk_{in}, \mathsf{Sign}(tx, sk_{in}))$ and send $(\textsc{SignTx}, \mathsf{sid}, stx)$ to $\mathcal{C}$.

4 Security in the Hybrid Setting

We now assess the security of our proposed model in order to prove the security of the hybrid setting with respect to the wallet ideal functionality. The attacks that our model considers are the following:

1) *Privacy loss*: when the adversary corrupts a client, he accesses the account's public keys, addresses and their balance;
2) *Payment attack*: during the transaction issuing operation, the adversary may tamper with the inputs to alter the payment amount, the receiving address, and/or the fee amount. This attack is successful if and only if the client is corrupted and the user deviates from her expected behavior, *i.e.*, does not reject the malicious transaction data;
3) *Address generation attack*: the adversary may tamper with address generation on the client's side, so that the user acquires an address which is adversarially controlled. This attack will be successful if and only if the client is corrupted and the user deviates from her expected behavior, *i.e.*, does not cross-check the address that the client provides with the hardware one;
4) *Chain attack*: the adversary may tamper with the balance calculation by providing to the wallet a malicious chain. This family of attacks is successful only if the client is corrupted, regardless if the user follows the protocol.

Chain Attacks The attacks 1, 2 and 3 have been previously identified by empirical studies showcasing their applicability [14,2]. However, the *chain attack* has not been previously discussed, and is more nuanced compared to the others. Under our model, the client is the only party that connects to the network. Therefore, a corrupted client can mount any type of eclipse attack [15], including the chain attacks that we describe here. We showcase an example of these attacks.

Assume the honest chain $\mathsf{chain}_{\mathsf{w}}$, and a transaction tx, which transfers θ_{in} funds to an address α and is published in the j-th block of $\mathsf{chain}_{\mathsf{w}}$. Prior to block j, *i.e.*, blocks with indices in $[0, j-1]$, a number of transactions were published that sent an aggregated amount of θ_{past} funds to α. The adversary $\mathcal{A}$ substitutes $\mathsf{chain}_{\mathsf{w}}$ with a chain $\mathsf{chain}_{\mathcal{A}}$, which is the prefix chain up to, but not including, the j-th block, *i.e.*, it consists of the blocks with indices $0 \dots j-1$. Hence, during the balance calculation, the wallet assumes that α owns θ_{past} funds. When the user requests a transaction $tx = (\alpha_r, \theta_{pay}, \theta_{fee})$, where $\theta_{past} = \theta_{pay} + \theta_{fee}$ (same as $\theta_{past} > \theta_{pay} + \theta_{fee}$), the wallet computes the amount of change according to θ_{past}, and spends the rest as fees. The attack results in forcing the wallet to spend more funds than it would in an honest setting.

The Hybrid Setting Security Theorem In order to prove the security of our model, we denote the hybrid setting described in Section 3.4 by π_{hybrid}. We show that π_{hybrid} securely realizes the wallet ideal functionality $\mathcal{F}_{\text{w}}$ defined in Section 3.3. In the ideal execution, $\mathcal{G}_{\mathsf{LEDGER}}$ uses the ideal wrapper IdealValidateWrapper defined in Section 3.2, whereas in the real world it utilizes RealValidateWrapper. Theorem 1 restricts to environments that do not corrupt the hardware party $\mathcal{H}$, therefore it cannot cover attacks mounted by the hardware wallet's manufacturer or cases when the hardware wallet gets corrupted due to insecure hardware.

Theorem 1 (Hybrid Wallet). *Let the hybrid setting π_{hybrid}, which is parameterized by a signature scheme Σ and a hash function* H *and interacts with $\mathcal{G}_{\mathsf{LEDGER}}$ parameterized by* RealValidateWrapper*. π_{hybrid} securely realizes the ideal functionality $\mathcal{F}_{\text{w}}$, which interacts with $\mathcal{G}_{\mathsf{LEDGER}}$ parameterized by* IdealValidateWrapper*, if and only if Σ is EUF-CMA and* H *is an instantiation of the random oracle.*

Proof. We refer to Appendix A for the proof. □

Theorem 1 can be used to prove the security of any wallet scheme that realizes the hybrid setting. To evaluate a wallet implementation, first it is identified whether it realizes the *human, client* and *hardware* protocols. Under the premise of a faithful realization of these protocols, *i.e.*, in terms of exchanged messages and internal operations, the security assumptions of its building components are evaluated. More precisely, the signature algorithm that the wallet uses must be EUF-CMA, the hash function must act as a random oracle, and the communication channels between the parties must be secure. Typical examples of such components are the *ECDSA* [18] signature algorithm and a *SHA-2* [23] hash function. If these assumptions hold, then the wallet is secure under our model.

The Negligent User In Section 3.4 we presented a well-defined protocol that the user should follow. As shown in Section 4, as long as the parties follow the defined protocols faithfully - and the cryptographic primitives used are strong enough - then the hardware wallet setting is secure. The integrity of the transaction issuing and the address generation operations are entirely based on the premise that the user will identify any malicious data, by comparing correctly the data shown by the client with the data shown by the hardware. However, even though this might be trivial for software, *e.g.*, for the client and the hardware wallet, people are prone to errors. Comparison of long hexadecimal strings has long been proved a challenging procedure, with many research outcomes suggesting that it is unrealistic to expect a perfect comparison of cryptographic hashes *e.g.*, [25,16,24], as humans find this process difficult and are prone to errors. In real world scenarios, the user aims at performing any operation quickly and being into a hurry often causes deviations from the expected behavior. Additionally, expecting the user to manually copy a Bitcoin address shown on the hardware's screen, defeats the usability purposes of the wallets. Thus, it is more than possible that the user will choose to simply copy the address directly from the client. However, such usability difficulties of the compare-and-confirm process open an attack vector for the payment and address generation attacks.

We model the probability of a user diverging from the human protocol π_{human} as a random variable $R_h \in [0, 1]$, which equally denotes the probability of successful payment and address attacks. The distribution of R_h varies, depending both on the vigilance of the user and usability parameters. For example, a user allowing all requests to be completed without checking, *i.e.*, because the process takes too long and the data is difficult to read, would be identified by R_h close to 1. A user who carefully checks the data, *i.e.*, because there are no time restrictions or because the hardware presents it is such way that captures the user's attention, would be identified by R_h closer to 0. Another factor that may affect R_h is the length of the addresses: the longer the address, the more difficult to read and compare. However, the experimental evaluation of R_h through usability studies of Bitcoin addresses and the user's capability to compare-and-confirm them correctly is out of the scope of this work and is left as future research.

5 Product evaluation

As of September 2018, the hardware wallets suggested by bitcoin.org are Digital Bitbox, KeepKey, Ledger, and Trezor. All, except Digital Bitbox, have an embedded screen to present information to the user, thus we focus on KeepKey, Trezor and Ledger. We manually inspected these wallets, extracted their protocols, and mapped them to our model. Our results show that the implementations bare significant similarities. Although the wallets do have different low-level implementations, the protocols that they execute are captured by the hybrid setting presented in Section 3.4. Instantiating our model to the actual implementations indicates the correctness of previous empirical studies, which suggest that the Ledger wallets are prone to the payment [14] and address generation [2] attacks. The wallets are subject to these attacks when the client is dishonest and are secure only if the cryptographic primitives are secure and the user does not deviate from the defined protocol, *i.e.*, successfully identifies any tampered data. Moreover, the instantiation of our model to the three implementations suggests that the wallets are prone to the chain attack, which has not been previously discussed. In this case, the attack cannot be blocked by the user, thus the wallets are secure against these attacks if and only if the underlying cryptographic mechanisms are secure *and* the client is honest.

In this section we use the model of Section 3.4 to evaluate these products. We identify whether such implementations are faithful to our protocols and, if not, identify the possible attacks that can be mounted against them. We expect this type of evaluation to become an industry standard for hardware wallets, so that vendors can improve the security and performance of their products by employing formal verification methods, instead of empirical techniques.

For each implementation we focus on the two core wallet operations: *address generation* and *transaction issuing*. Since all implementations are susceptible to chain attacks, we focus on the viability of payment and address attacks in each case. We show that Trezor and KeepKey are secure against payment and address

attacks, as long as the user follows the protocol and verifies the data, whereas Ledger wallets are prone to address attacks, due to divergence from our model.

Trezor and KeepKey We investigate the implementation of the Trezor Model T and KeepKey hardware wallets. Both products are implemented similarly, so we will focus Trezor, since our findings also apply to KeepKey. Trezor provides a touch screen for both displaying information and receiving input from the user. Based on the developer's guide[3], which is publicly accessible, we describe an abstraction of Trezor's behavior under our model.

During address generation, Trezor requires that the user connects the token to the client and unlocks it, *i.e.*, the user *initiates a session* similar to our model definition. The client then retrieves the address from the hardware token and displays it to the user. The hardware also displays the address, as long as the "Show on Trezor" option is enabled[4]. If this option is disabled, then the user cannot verify the client's address and is prone to an address attack, *i.e.*, the client might display a malicious address which the user cannot cross-check with the hardware wallet. However, the user manual does urge the user to always check the two addresses[5], in order to avoid such attack scenarios.

During transaction issuing, the user again connects the device to the client and unlocks it. Then she initiates a transaction by giving to the client the recipient's address, and the payment and fee amounts, similarly to our hybrid model setting. The client initiates the transaction signing process with the hardware by providing this data, which the token then displays to the user for verification[6]. After the user has verified the transaction, the hardware communicates with the client and signs the needed data[7]. Again, given our high level investigation, this process matches the communication steps that our model describes.

Ledger We investigate the implementation of Ledger Nano S according to the user manual[8] and our own analysis. Similarly to Trezor, before performing any operation the user is required to initiate a session by connecting the hardware to the client and unlocking it. The hardware provides a small screen for displaying information and a pair of two buttons for receiving commands from the user.

During the address generation, the client displays the newly generated address to the user. However, there is no option for the hardware wallet to also display the address[9], so that the user can cross-check and verify the two. This is a clear divergence from our model and allows for address attacks, *e.g.*, by a corrupted client that displays a malicious address to the user.

[3] Trezor developer's guide: `https://wiki.trezor.io/Developers_guide`
[4] See: `https://wiki.trezor.io/Developers_guide:Trezor_Connect_API_Methods`
[5] See: `https://wiki.trezor.io/User_manual:Receiving_payments`
[6] See: `https://wiki.trezor.io/User_manual:Making_payments`
[7] See: `https://wiki.trezor.io/Developers_guide:API_Workflows`
[8] See: `https://support.ledgerwallet.com/hc/en-us/articles/360009676633`
[9] Ledger has issued firmware update to address this issue and allow both the client and the hardware to generate and display the address. However, the firmware needs to be updated manually, a process that is commonly neglected by common users.

The transaction issuing process is also similar to Trezor and captured by our model: the user inputs to the client the transaction data, *i.e.*, the recipient's address, and the payment and the fee amounts. The client forwards this data to the hardware, which displays it to the user for verification. After receiving the user's confirmation, the hardware interacts with the client in order to sign and publish the transaction.

6 Conclusion

The presented work is the first effort to formally describe Bitcoin wallets. We focus on hardware wallets, as they are considered the most secure means of account management, while also being the least studied part of cryptocurrency ecosystems, and devise a model to formally prove their security specifications. We prove that their security is not one-dimensional and entirely based on secure primitives as expected; external factors such as the client to which the hardware connects and the user who operates the wallet play a major role in the overall wallet's security. Our model provides a guide for implementing and verifying existing or future wallets. Indeed, by evaluating the Keepkey, Ledger and Trezor wallets we show that security can only be guaranteed if the cryptographic primitives are secure *and* if each party executes their protocol correctly. However, since a user's deviation from the protocol is to be expected, due to human errors and usability problems of hash comparison techniques, future work will focus on evaluating this error probability and proposing techniques to reduce such risk.

Acknowledgements

This work was partially supported by the EPSRC grant EP/P002692/1.

References

1. KeepKey. `https://keepkey.com/` (2018), [Online; accessed 1-Sep-2018]
2. Ledger Receive Attack. `https://www.docdroid.net/Jug5LX3/ledger-receive-address-attack.pdf` (2018), [Online; accessed 19-Sep-2018]
3. Trezor. `https://trezor.io/` (2018), [Online; accessed 1-Sep-2018]
4. Alois, J.: Ethereum parity hack may impact eth 500.000 or 146 million (2017)
5. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of bitcoin transactions. Financial Cryptography and Data Security. LNCS, Springer (2018)
6. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. pp. 324–356 (2017)
7. Bamert, T., Decker, C., Wattenhofer, R., Welten, S.: Bluewallet: The secure bitcoin wallet. In: International Workshop on Security and Trust Management. pp. 65–80. Springer (2014)
8. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 104–121. IEEE (2015)

9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. pp. 136–145 (2001)
10. Canetti, R.: Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239 (2003), http://eprint.iacr.org/2003/239
11. Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. Cryptology ePrint Archive, Report 2002/059 (2002), http://eprint.iacr.org/2002/059
12. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 281–310. Springer (2015)
13. Gentilal, M., Martins, P., Sousa, L.: Trustzone-backed bitcoin wallet. In: Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems. pp. 25–28. ACM (2017)
14. Gkaniatsou, A., Arapinis, M., Kiayias, A.: Low-level attacks in bitcoin wallets. In: International Conference on Information Security. pp. 233–253. Springer (2017)
15. Heilman, E., Kendler, A., Zohar, A.: Eclipse attacks on bitcoin's peer-to-peer network.
16. Hsiao, H.C., Lin, Y.H., Studer, A., Studer, C., Wang, K.H., Kikuchi, H., Perrig, A., Sun, H.M., Yang, B.Y.: A study of user-friendly hash comparison schemes. In: Computer Security Applications Conference, 2009. ACSAC'09. Annual. pp. 105–114. IEEE (2009)
17. Huang, D.Y., Dharmdasani, H., Meiklejohn, S., Dave, V., Grier, C., McCoy, D., Savage, S., Weaver, N., Snoeren, A.C., Levchenko, K.: Botcoin: Monetizing stolen cycles. In: NDSS. Citeseer (2014)
18. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**(1), 36–63 (2001)
19. Lim, I.K., Kim, Y.H., Lee, J.G., Lee, J.P., Nam-Gung, H., Lee, J.K.: The analysis and countermeasures on security breach of bitcoin. In: International Conference on Computational Science and Its Applications. pp. 720–732. Springer (2014)
20. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
21. Parker, L.: Bitcoin stealing malware evolves again. https://bravenewcoin.com/news/bitcoin-stealing-malware-evolves-again/ (2016), [Online; accessed 1-Sep-2018]
22. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 643–673. Springer (2017)
23. Penard, W., van Werkhoven, T.: On the secure hash algorithm family. Cryptography in Context pp. 1–18 (2008)
24. Tan, J., Bauer, L., Bonneau, J., Cranor, L.F., Thomas, J., Ur, B.: Can unicorns help users compare crypto key fingerprints? In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. pp. 3787–3798. ACM (2017)
25. Uzun, E., Karvonen, K., Asokan, N.: Usability analysis of secure pairing methods. In: International Conference on Financial Cryptography and Data Security. pp. 307–324. Springer (2007)
26. Vasek, M., Bonneau, J., Ryan Castellucci, C.K., Moore, T.: The bitcoin brain drain: a short paper on the use and abuse of bitcoin brain wallets. Financial Cryptography and Data Security, Lecture Notes in Computer Science. Springer (2016)
27. Volotikin, S.: Software attacks on hardware wallets. Black Hat USA 2018 (2018)
28. Wuille, P.: Hierarchical Deterministic Wallets. https://en.bitcoin.it/wiki/BIP_0032 (2018), [Online; accessed 1-Sep-2018]

A Security in the Hybrid Setting

Theorem 1 (Hybrid Wallet). *Let the hybrid setting π_{hybrid}, which is parameterized by a signature scheme Σ and a hash function* H *and interacts with* $\mathcal{G}_{\mathsf{LEDGER}}$ *parameterized by* RealValidateWrapper*. π_{hybrid} securely realizes the ideal functionality $\mathcal{F}_{\mathrm{w}}$, which interacts with $\mathcal{G}_{\mathsf{LEDGER}}$ parameterized by* IdealValidateWrapper*, if and only if Σ is EUF-CMA and* H *is an instantiation of the random oracle.*

Proof. **The "if" part** For this part of the theorem we assume that the environment $\mathcal{Z}$ can distinguish between the ideal and the real execution with non-negligible probability. We then describe a "generic" simulator S for each adversary $\mathcal{A}$, which emulates the interfaces defined by the functionality. S also runs an internal copy of $\mathcal{A}$ and forwards the outputs of its computations to $\mathcal{A}$. We then construct a forger G that runs an internal simulation of the environment $\mathcal{Z}$. Thus, for each property assumption, we show that there exists a "bad" event E such that, as long as E does not occur, the two executions are statistically close. However, when E occurs, the environment $\mathcal{Z}$ distinguishes between the executions. At this point, G uses $\mathcal{Z}$ and outputs the values that break the property under question. Therefore since, by assumption, E occurs with non-negligible probability, we show that G is also succesful with non-negligible probability.

The simulator Let us now construct the generic simulator S. For every interface defined by the ideal functionality, S completes the operations in the manner defined by the protocols in the hybrid setting. It internally runs a copy of the adversary $\mathcal{A}$ and forwards the necessary messages to it as defined in the hybrid setting. So, the view of the $\mathcal{A}$ when it interacts with S is the same as in the case it operates in the real world setting. S performs as follows:

- Any inputs received from the environment $\mathcal{Z}$, forward them to the internal copy of $\mathcal{A}$. Moreover, forward any output from $\mathcal{A}$ to $\mathcal{Z}$;
- **Party Setup:** For every party P for which $\mathcal{F}_{\mathrm{w}}$ sends messages, spawn an internal simulation of the parties for human $\mathcal{U}$, client $\mathcal{C}$ and hardware wallet $\mathcal{H}$, which also interact with $\mathcal{A}$ as needed and run the protocols π_{human}, π_{client} and π_{hw} respectively;
- **Party Corruption:** Whenever the adversary $\mathcal{A}$ corrupts a party, S corrupts it in the ideal process and hands to $\mathcal{A}$ its internal state;
- **(Setup, Initialize Session, Generate Address, Issue Transaction):** For any message for these interfaces, follow the protocols π_{human}, π_{client} and π_{hw} for the human, client and hardware parties.

In order to prove the theorem regarding the properties of the signature scheme we follow the reasoning of Canetti [10]. We will show the proof for the *unforgeability* property of the signature scheme, as the proofs for the other properties are similar to it.

Unforgeability: Assume that *consistency* and *completeness* hold for Σ and H instantiates the random oracle. In this case, the *Setup, Initialize Session* and *Generate Address* interfaces are the same in the both settings from the adversary's point of view. Since, by assumption, $\mathcal{Z}$ distinguishes between the two, this

occurs during the *Issue Transaction* phase, *i.e.*, by observing a valid signature of a transaction which has not been issued by the hardware wallet.

We now construct a forger G that runs a simulated copy of $\mathcal{Z}$. G follows the generic simulator as above, except for the transaction issuing interface. Upon receiving $(\textsc{Submit}, \mathsf{sid}, stx)$, where $stx = (tx, vk, \sigma)$, it checks if $\mathsf{Verify}(tx, vk, \sigma) = True$. If so, it accesses the internal state of the hardware $\mathcal{H}$ and checks whether it has issued stx. If so, then it continues the simulation. Else G outputs stx as a forgery. Since, as long as this does not occur, the two executions are statistically close and, by assumption, $\mathcal{Z}$ is succesful with non-negligible probability, then the probability that G is also succesful is non-negligible.

The "only if" direction We show that if one property does not hold, then the probability that the "bad" event E (as above) occurs is non-negligible, so that the environment $\mathcal{Z}$ can distinguish between the real and ideal executions.

Again we prove the theorem for the *unforgeability* property - the proofs for the other properties of Σ are constructed similarly.

Unforgeability: Assume that *unforgeability* does not hold for Σ, so there exists a forger G for Σ. When G wishes to obtain a signature for some message m, the environment sends the message $(\textsc{IssueTx}, \mathsf{sid}, m)$ and forwards the response to G. When G outputs a forgery $stx = (tx, vk, \sigma)$, if tx has been previously signed then the environment halts. Else it sends stx to $\mathcal{G}_{\mathsf{LEDGER}}$ and observes the ledger's updates. In the ideal setting the transaction will be rejected by the validation predicate and it will never be included in the ledger, whereas in the real world the probability that the transaction is accepted and eventually published in the ledger is non-negligible.

Finally, we show the proof for the *address randomness* property which accompanies the assumption that H instantiates a random oracle.

Address randomness: Assume that all properties for Σ hold. Now the *Setup, Initialize Session* and *Issue Transaction* interfaces are similar in both settings. So if $\mathcal{Z}$ distinguishes between the two worlds, then this occurs during an address generation interaction. Specifically, it should observe addresses which are not uniformly distributed over the space of possible addresses. This is impossible in the ideal world by construction. However, if this was true for the real world, then H would not instantiate the random oracle, therefore by assumption it is impossible for $\mathcal{Z}$ to distinguish between the two worlds. □